How to Easily Concatenate Two Pandas DataFrames

Authored by

stats writer

December 2, 2025

The ability to efficiently combine datasets is fundamental to data analysis using the Pandas library in Python. When working with tabular data structures, specifically the Pandas DataFrame, analysts frequently encounter scenarios where multiple files or segmented results need to be unified. Pandas offers a powerful and flexible tool for this purpose: the concat function.

The primary role of the `pd.concat()` function is to take a list of Pandas DataFrame objects and stack them together along a specified axis. Unlike SQL-style merging, which combines data based on common keys (e.g., using `pd.merge()`), concatenation is about aligning structures either vertically (stacking rows) or horizontally (adding columns side-by-side). Understanding the difference between these two primary operations--stacking versus joining--is crucial for effective data manipulation.

When you call the concat function, it requires an iterable object, typically a list, containing the DataFrames you wish to combine. The resulting Pandas DataFrame will seamlessly integrate the data from all inputs. Furthermore, the behavior of the concatenation is controlled by the `axis` parameter. By default, `axis=0`, which results in vertical stacking (combining rows). Setting `axis=1` performs horizontal concatenation (combining columns).

To perform a basic vertical concatenation, stacking DataFrames `df1` and `df2`, you can use the following syntax. This method is exceptionally useful when appending new time-series data or unifying datasets that share identical column structures:

df3 = pd.concat(, ignore_index=True)

This powerful syntax not only combines the data but also includes the optional argument `ignore_index=True`, which is essential for creating a clean, sequential index in the resulting DataFrame. The following sections will provide a detailed, practical demonstration of this functionality, illustrating its impact on data structure and indexing.

Example 1: Performing Basic Vertical Concatenation (Row Stacking)

Vertical concatenation, achieved when the `axis` argument defaults to 0, is the most common use case for the concat function. This operation involves placing the rows of the second DataFrame directly below the rows of the first. This is ideal when your data is split across multiple files but represents the same underlying schema (i.e., the columns are identical).

For this demonstration, we define two simple Pandas DataFrame objects, `df1` and `df2`, representing basketball statistics for two distinct teams, 'A' and 'B'. Both DataFrames possess the same three columns: 'team', 'assists', and 'points'. Our goal is to unify these two distinct datasets into a single analytical table.

The initial setup requires importing the Pandas library and defining our foundational data structures. Note how the column names and data types are consistent across both DataFrames, which is a prerequisite for seamless vertical stacking. The initial view of the DataFrames reveals their independent structure before unification.

import pandas as pd

# Define DataFrames
df1 = pd.DataFrame({'team': ,
'assists': ,
'points': })

df2 = pd.DataFrame({'team': ,
'assists': ,
'points': })
# View DataFrames prior to concatenation
print(df1)

team assists points
0 A 5 11
1 A 7 8
2 A 7 10
3 A 9 6

print(df2)

team assists points
0 B 4 14
1 B 4 11
2 B 3 7
3 B 7 6

To concatenate these two structures, we simply pass them as a list to the `pd.concat()` function, omitting the `axis` parameter since the default value (0) performs the desired vertical stacking. The resulting DataFrame, `df3`, now contains all 8 rows of data combined from the two input sources.

 # Concatenate the DataFrames using default axis=0
df3 = pd.concat()

# View resulting DataFrame
print(df3)

team assists points
0 A 5 11
1 A 7 8
2 A 7 10
3 A 9 6
0 B 4 14
1 B 4 11
2 B 3 7
3 B 7 6

As demonstrated in the output above, the resulting Pandas DataFrame successfully incorporates all rows from both inputs. However, a critical detail emerges regarding the row indices. Notice that the indices are preserved from the original DataFrames, leading to duplicate index labels (0, 1, 2, 3 appear twice). While Pandas handles these duplicates internally, they can complicate subsequent indexing operations or data retrieval tasks, making the index unreliable as a unique row identifier.

Managing Indices with the `ignore_index` Argument

When performing vertical concatenation, maintaining the original index structure often leads to the issue of non-unique index labels, as shown in the previous example. For most analytical workflows, particularly when creating a finalized dataset, it is necessary to reset or ignore the original indices to create a clean, sequential, zero-based index for the unified DataFrame. This is where the boolean argument `ignore_index` becomes invaluable.

Setting `ignore_index=True` instructs the concat function to discard the indices of the input DataFrames and automatically generate a brand new, numerically sequential index for the final combined result. This ensures that every row has a unique index label, simplifying data access and maintenance.

We modify the previous syntax by adding the `ignore_index=True` parameter to the `pd.concat()` call. This small but significant adjustment guarantees the integrity of our resulting index. It is generally considered best practice to utilize this argument unless there is a specific analytical need to preserve the source indices (e.g., if the index represents unique, meaningful keys like timestamps or identifiers).

 # Concatenate the DataFrames and ignore index
df3 = pd.concat(, ignore_index=True)

# View resulting DataFrame
print(df3)

team assists points
0 A 5 11
1 A 7 8
2 A 7 10
3 A 9 6
4 B 4 14
5 B 4 11
6 B 3 7
7 B 7 6

Observing the output, we can confirm the successful implementation of the index reset. The index of the combined DataFrame now ranges sequentially from 0 up to 7, precisely covering all eight rows of data. This streamlined indexing structure facilitates efficient data slicing, locating specific rows, and preparing the DataFrame for subsequent analytical stages.

Example 2: Horizontal Concatenation (Column Adding) using `axis=1`

While vertical stacking is common, the concat function is equally adept at combining DataFrames horizontally, effectively adding columns side-by-side. This operation is invoked by setting the `axis` parameter to 1 (`axis=1`). When concatenating horizontally, Pandas aligns the DataFrames based on their indices. If the indices match perfectly, the result is a full combination of columns.

Consider a scenario where `df1` contains initial player statistics and `df4` contains supplementary information (like player height or weight) for the exact same set of players, meaning their row indices correspond exactly. If we concatenate these two DataFrames with `axis=1`, the columns of `df4` will be appended directly to the right of `df1`'s columns.

It is vital to understand that horizontal concatenation relies heavily on index alignment. If the input DataFrames have differing indices, Pandas will introduce missing values (NaN) for rows where no matching index is found in the counterpart DataFrame. This behavior is similar to an outer join in SQL, ensuring all data is preserved, but potentially introducing sparsity.

The flexibility of `pd.concat()` allows it to handle data merging in ways that complement the more relational-database-oriented `pd.merge()`. While `merge()` is used for combining datasets based on key columns, `concat()` is often used for structural combination where index alignment is the primary criterion for joining data horizontally.

Advanced Control: The `join` Parameter and Data Alignment

When concatenating DataFrames that do not have perfectly overlapping columns (for vertical

concatenation) or perfectly matching indices (for horizontal concatenation), Pandas needs to determine which data to keep. This behavior is controlled by the `join` parameter, which defaults to `'outer'` when not specified.

`join='outer'` (Default): This performs a union of the index levels (or column names). The resulting DataFrame will contain all indices (or columns) present in any of the input DataFrames. Non-matching positions are filled with NaN values. This is suitable when you want to retain all available data, even if it leads to sparse rows or columns.

`join='inner'` : This performs an intersection of the index levels (or column names). The resulting DataFrame will only contain indices (or columns) that are common across *all* input DataFrames. Any data unique to a single input is discarded. This is useful when strict structural consistency is required across the combined dataset.

For example, if you vertically concatenate two DataFrames where `df1` has columns A, B, C and `df2` has columns B, C, D, an outer join will result in a combined DataFrame with columns A, B, C, D, using NaN where data is missing. An inner join will only result in columns B and C.

Using `keys` to Create a Hierarchical Index (MultiIndex)

A powerful feature of the concat function is the ability to easily track the source of the concatenated data using the `keys` parameter. By providing a list of strings to the `keys` parameter (matching the order of the DataFrames in the list), Pandas automatically creates a Pandas DataFrame with a hierarchical index, also known as a MultiIndex.

This approach is significantly beneficial for data provenance and subsequent subgroup analysis. The top level of the index will be populated by the keys provided, clearly segmenting the data based on its original DataFrame. This eliminates the need to manually add a source column before concatenation.

For instance, if we concatenate `df1` and `df2` using `keys=`, the resulting DataFrame will have an index structure where the first level indicates whether the row originated from 'Source_1' or 'Source_2', followed by the original index. This preserves the original index context while providing a clear grouping label.

Conclusion and Best Practices for Concatenation

The `pd.concat()` function is an indispensable tool for combining DataFrames in Pandas, offering versatility for both vertical stacking (appending rows) and horizontal joining (adding columns). Mastering its key arguments ensures accurate and reliable data assembly for analysis.

To summarize the critical parameters and best practices:

List Input: Always pass the DataFrames to be combined as a Pandas DataFrame list, e.g., `pd.concat()`.

Axis Control: Use `axis=0` for stacking rows (default) and `axis=1` for adding columns side-by-side.

Index Management: Use `ignore_index=True` when vertical concatenation leads to duplicate indices and you require a fresh, sequential index.

Alignment Control: Use the `join` parameter (`'outer'` or `'inner'`) to specify how to handle mismatched columns or indices during the concatenation process.

By applying these techniques, data scientists can efficiently preprocess large volumes of segmented data, transforming disparate sources into a cohesive and robust analytical framework, ensuring the integrity and usability of the combined dataset.

Note #1: As previously mentioned, while this demonstration focused on two DataFrames, you can use this exact syntax to concatenate any number of DataFrames simultaneously, provided they are all included in the input list.

Note #2: For comprehensive details on all parameters, including `verify_integrity` and the nuances of the `keys` argument, you can find the complete documentation for the pandas **concat()** function here.